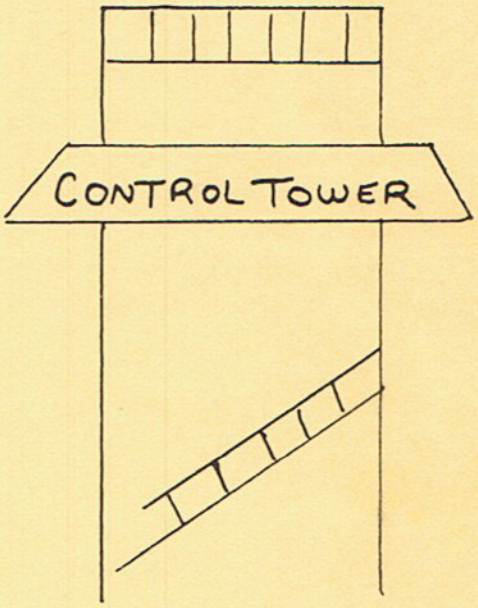
TIS Workbook 6

TOTAL INFORMATION SERVICES



PET Control and Logic Statements

PET CONTROL AND LOGIC STATEMENTS

Table of Contents

I. INTRODUCTION

This workbook gives a series of exercises that show the new user how to use the control statements for selection, branching and subroutines on your Commodore PET 2001 computer. The most effective way to use the workbook is to sit down with a PET and go through the exercises as they are presented. Enough space has been provided in the workbook for you to add your own examples as you develop them. Later, when you need to refresh your memory on a particular topic, these examples should supply pertinent, meaningful information.

A. Assumptions Made About the User

Some PET users are comfortable with mathematics. However, this workbook assumes that the majority is not. For that matter, most exercises will use nothing more than high school arithmetic.

This workbook also assumes you know something about the syntax and semantics of the BASIC programming language. If you do not, we recommend that you obtain one of the following books:

BASIC Programming, J. Kemeny and T. Kurtz
BASIC, Albrecht, Finke, Brown
What Do You Do After You Hit Return? People's Computer Co.
Basic BASIC, James Coan
Advanced BASIC, James Coan

Read about BASIC syntax; then alternate between this workbook and your text on BASIC. That way you will learn both BASIC and how to use the PET.

B. Notation Used in This Workbook

We use a consistent notation in this workbook to indicate what is to be typed on the keyboard (T:), what appears on the TV display (R:), and what indicates blanks are to be typed (b). For example:

T: info ('RETURN') key

means to type the characters contained on the line after the colon (:) followed by a 'RETURN'.

R: response

means that the system response to the previous T: line should be a line on the TV.

Blanks are important. They are specified by b. For example:

T: ?"ABbC" ('RETURN' key)

means type ?, then ", then the letter A, then the letter B, then

a space, then the letter C, then " followed by a 'RETURN'. Now let's run that example all together:

```
T:  ?"ABbC" ('RETURN' key)
R:  AB C
```

The special keys on the PET keyboard can cause some confusion. This workbook identifies the special keys with the notation 'KEYNAME'. 'KEYNAME' means press the named key. The unquoted sequence of characters OTHER means press the five keys O T H E R in succession.

Example: Special key notation

```
T:  'STOP'
```

means press the key labeled STOP.

```
T:  STOP
R:  BREAK
R:  READY.
```

means press the four keys S T O P in succession.

II. BRANCHING

The PET BASIC has two statements for branching. The GO TO statement is an unconditional one-way branch to a single specified statement number. The ON X GO TO statement is an unconditional multi-way branch to one of several statement numbers. The particular statement chosen is specified by X and the list of possible statement numbers.

A. GO TO

An unconditional branch (transfer of control) to a single statement is written as GO TO nnn where nnn is a statement number. The statement number must be a numeric constant. Variables or string constants are not allowed.

Exercise: Check for legal statement numbers in the GO TO statement.

```
T:  NEW
T:  100 PRINT "GO TO 100":END
T:  GO TO 100
R:  GO TO 100

T:  GO TO "100"
R:  UNDEF'D STATEMENT ERROR

T:  S=100:GO TO S
R:  UNDEF'D STATEMENT ERROR

T:  S$="100":GO TO S$
R:  UNDEF'D STATEMENT ERROR
```

You can see that a numeric constant is the only legal way to specify the destination of a GO TO.

The statement number that you want to GO TO must exist or an error message will be given.

```
T:  GO TO 110
R:  UNDEF'D STATEMENT ERROR
```

B. ON X GO TO

The multi-way branch is written ON X GO TO n1, n2, . . .nn where X is the variable that selects which branch to take and the n's are the possible statements to branch to. The destination is the xth n. For example, if X is 2, then control is transferred to the

2nd statement number in the list (n2). If X is 4, control is transferred to the 4th statement in the list (n4).

Exercise: Show how ON X GO TO works.

```
T:  NEW
T:  1 PRINT "STATEMENT #1":STOP
T:  2 PRINT "STATEMENT #2":STOP
T:  3 PRINT "STATEMENT #3":STOP
T:  4 PRINT "STATEMENT #4":STOP
T:  5 PRINT "STATEMENT #5":STOP
T:  100 I=2
T:  110 ON I GO TO 1,2,3,5
T:  120 PRINT "SLIPPED BY GO TO"
T:  RUN 100
R:  STATEMENT #2
R:  BREAK IN 2
```

If you get STATEMENT #1 and BREAK IN 1, you probablytyped just RUN and not RUN 100. Try RUN 100.

```
T:  100 I=4
T:  RUN 100
R:  STATEMENT #5
R:  BREAK IN 5
```

Notice that the ON X went to the 4th statement number in the list and NOT to statement number 4.

Exercise: Show how ON X handles values outside the legal range.

```
T:  100 I=0
T:  RUN 100
R:  SLIPPED BY GO TO

T:  100 I=5
T:  RUN 100

R:  SLIPPED BY GO TO
```

If the variable is outside the quantity of statement numbers provided, the next statement in sequence is executed.

Exercise: Show the legal range of values for the variable I in ON I ... .

```
T:  100 I=255
R:  RUN 100
R:  SLIPPED BY GO TO

T:  100 I=256
T:  RUN 100
R:  ?ILLEGAL QUANTITY ERROR IN 110

T:  100 I=-1
T:  RUN 100
R:  ?ILLEGAL QUANTITY ERROR IN 110
```

This shows that the selector value must be between 0 and 255.

Exercise: Use ON X GO TO to select one of four actions. Let's say that you are writing a program to prepare a mailing list and that there are four actions to take.

1. Add a new name to the list.
2. Change an existing name.
3. Print out the list.
4. Count the number of entries on the list.

You could set up the program to accept as input the numbers 1-4 to select the action.

```
T:  NEW
T:  100 PRINT "MAILING LIST SKELETON"
T:  110 INPUT "ENTER ACTION #";A
T:  130 ON A GO TO 1000, 2000, 3000, 4000
```

The choice of statement numbers is somewhat arbitrary. You can make your programs more readable by developing a consistent pattern (style). It is easy to see that for A=n the statement to go to is n000.

```
T:  150 GO TO 110
T:  1000 PRINT "ADD SELECTED"
T:  1990 GO TO 150
T:  2000 PRINT "CHANGE SELECTED"
T:  2990 GO TO 150
T:  3000 PRINT "PRINT LIST REQUESTED"
T:  3990 GO TO 150
T:  4000 PRINT "COUNT REQUESTED"
T:  4990 GO TO 150
```

For each possible action number a stub is provided. The stub just tells you whether the control portion of the program is working

correctly or not. Later the PRINT statements can be replaced with the program steps to do the task described.

It may seem strange to have statement 150 (the GO TO 110) and each stub end with a GO TO 150. A "GO TO" another "GO TO" sounds very inefficient. It is inefficient. A GO TO directly is a little faster than a "GO TO" another "GO TO". However, if you use this approach of always returning to the statement after the ON X GO TO statement you won't have to guess where the program is going. That makes debugging much easier.

Now try all the legal inputs.

```
T:   RUN
R:   MAILING LIST SKELETON
R:   ENTER ACTION #?
T:   1
R:   ADD SELECTED
R:   ENTER ACTION #?

T:   2
R:   CHANGE SELECTED
R:   ENTER ACTION #?

T:   3
R:   PRINT LIST REQUESTED
R:   ENTER ACTION #?

T:   4
R:   COUNT REQUESTED
R:   ENTER ACTION #?
```

To terminate this program type 'STOP' and 'RETURN' Save this program as MAIL1. See WB-4, PET Cassette for the proper save and verify sequence. We will use MAIL1 later and add features to it.

Exercise: Check how MAIL1 reacts to errors in input. If you have turned power off reload MAIL1.

```
T:   RUN
R:   MAILING LIST SKELETON
R:   ENTER ACTION #?
T:   0
R:   ENTER ACTION #?
T:   5
R:   ENTER ACTION #?
```

The program does not handle bad input very well.

You can use the fact that some out-of-range numbers GO TO the next statement to improve error handling.

```
T:   140 PRINT "INPUT MUST BE 1-4"

T:   RUN
R:   MAILING LIST SKELETON
R:   ENTER ACTION #?
T:   0
R:   INPUT MUST BE 1-4
R:   ENTER ACTION #?
```

Save this version of the program as MAIL2. Later we will use it as a base to make a more human-oriented interface. Since the PET can handle character strings as well as numbers, there is no reason that MAIL can't respond to requests specified by characters. In the section on IF, another improvement will be made to handle negative and large numbers for the value of A.

III. THE IF STATEMENT

The IF statement allows a program to do different things based on input parameters or the current value of a calculated value.

A. THEN form

The most general form of the IF statement is:

IF expression THEN statement.
When the "expression" is true, i.e., evaluates to non zero, the "statement" is executed next. If the "expression" is false, i.e., evaluates to zero, the "statement" is skipped and the following numbered line is executed next.

Exercise: Use the THEN form of the IF statement.

```
T:   NEW
T:   100 A=1
T:   110 IF A=1 THEN PRINT "A=1"
T:   120 PRINT "LINE 120"

T:   RUN
R:   A=1
R:   LINE 120

T:   100 A=2
T:   RUN
R:   LINE 120
```

B. GO TO Form

The IF statement does not have to have a THEN. The form:
IF expression GO TO nn is legal, it it the same as:
IF expression THEN GO TO nn. A third way to specify the same operation is:
IF expression THEN nn.

All three forms give the same result. If the expression is true continue execution at line nn. If the expression is false, continue execution at the next numbered line.

Exercise: Try the GO TO form of the IF statement. Use part of the program already entered.

```
T:   130 STOP
T:   140 PRINT "A=1"
T:   150 STOP
T:   110 IF A=1 GO TO 140
T:   RUN
R:   LINE 120
R:   BREAK IN 130
```

Since A=2, the IF test failed. This caused statements 120 and 130 to be executed.

```
T:  100 A=1
T:  RUN
R:  A=1
R:  BREAK IN 150
```

Here A=1 and statements 140 and 150 were executed.

Exercise: Use the short form of the GO TO in an IF statement.

```
T:  110 IF A=1 THEN 140
T:  RUN
R:  A=1
R:  BREAK IN 150

T:  100 A=2
T:  RUN
R:  LINE 120
R:  BREAK IN 130
```

These two runs show the same results as the previous exercise.

C. Multiple Statements

The THEN form of the IF statement can have multiple statements after the THEN. These multiple statements are executed if the test is true.

Exercise: Show how multiple statements after the THEN are handled.

```
T:  NEW
T:  100 A=1
T:  110 IF A=1 THEN PRINT"A=1":A=2:PRINT"A IS NOW=";A
T:  120 PRINT "LINE 120"
T:  130 STOP
T:  RUN

R:  A=1
R:  A IS NOW=2
R:  LINE 120
```

When the IF test is true, all the statements after the THEN are executed.

```
T:  110 IF A=1 THEN PRINT"A=1":GO TO 130:PRINT"NEVER HERE!"
T:  RUN
R:  A=1
R:  BREAK IN 130
```

The statement after GO TO 130 can never be executed. The GO TO prevents the PRINT "NEVER HERE!" from being reached in sequence. Since that PRINT does not have a statement (line) number, it cannot be reached by another GO TO. Since only one line number is permitted per line, you cannot insert a line number before the PRINT.

Exercise: Show that the second line number on a line cannot be reached.

```
T:  110 IF A=1 THEN PRINT"A=1":GO TO 130:115 PRINT"NEVER HERE!"
T:  RUN
R:  A=1
R:  BREAK IN 130

T:  GO TO 115
R:  ?UNDEF'D STATEMENT ERROR
```

This shows that statement number 115 cannot be found. Only the regular statement number on a line can be reached.

The following two lines will catch bad input and give the program a chance to continue.

```
T:  110 IF A=1 THEN PRINT "A=1":115 PRINT "NEVER HERE"
T:  RUN
R:  A=1
R:  ?SYNTAX ERROR IN 110
```

You can use what you have learned about the IF statement to improve the MAIL2 Program.

Exercise: Add error checking to MAIL2. Reload MAIL2. To refresh your memory on how this program handles erroneous input, try:

```
T:  RUN
R:  MAILING LIST SKELETON
R:  ENTER ACTION #
T:  -1
R:  ?ILLEGAL QUANTITY ERROR IN 130

T:  RUN
R:  MAILING LIST SKELETON
R:  ENTER ACTION #
T:  256
R:  ?ILLEGAL QUANTITY ERROR IN 130
```

```
T:  120 IF A<0 THEN 140
T:  122 IF A>255 THEN 140
T:  RUN
R:  MAILING LIST SKELETON
R:  ENTER ACTION #
T:  -5
R:  INPUT MUST BE 1-4
R:  ENTER ACTION #
T:  256
R:  INPUT MUST BE 1-4
R:  ENTER ACTION #
```

The results show that the program no longer terminates when bad input is typed. Save this program as MAIL3.

Exercise: Make MAIL3 more human-oriented. Change the input to accept the strings "ADD", "CHANGE", "PRINT", and "COUNT" instead of the numbers 1 to 4. Load MAIL3. A complete listing of the character oriented mailing list skeleton is in the Appendix.

```
T:  102 AT$(1)="ADD"
T:  104 AT$(2)="CHANGE"
T:  106 AT$(3)="PRINT"
T:  108 AT$(4)="COUNT"
```

This sets up the array AT$ with the four action words. You could put any string you desired in here.

```
T:  110 INPUT "ENTER ACTION":A$
T:  115 FOR A=1TO4
T:  120 IF A$<>AT$(A)THEN 135
T:  122
T:  135 NEXT A
```

Get the action request and search for a match.

```
T:  140 PRINT "INPUT MUST BE LEGAL ACTION"
```

Change error message to reflect what really is wrong.

```
T:  RUN
R:  MAILING LIST SKELETON
R:  ENTER ACTION?
T:  ADD
R:  ADD SELECTED
R:  ENTER ACTION?
T:
R:
```

Try a few legal and illegal command strings to convince yourself that the program works. Why don't you have to test explicitly for a value of A<0 OR A>255? Save this version of the mailing list skeleton as MAIL4.

IV. SUBROUTINES

As you become more experienced in programming you will probably write bigger and more complicated programs. Subroutines can help you get these programs to work more quickly and easily. Subroutines are useful to avoid repeated sections of almost identical statements, to simplify the organization of the program, to make the program easier to read and follow, and to make it easier to debug.

Once a subroutine has been debugged, you can use it as a building block in other programs. This is particularly useful if you can be sure that this building block does not conflict with or interfere with other building blocks. Section VI gives some hints on how to construct your subroutines so that they won't interfere with each other.

To use a subroutine you must keep in mind two principles. The first is linkage. When you want to use a subroutine, your program has to call (invoke) it. When the subroutine is finished, it must return to where it was called. Figure 4-1 shows this linkage pictorially. Subroutines are most useful if they can be called from several different places in your program. For this reason, you should try to put into a subroutine a function that you need done frequently and in several different places.

The second principle of subroutines is parameter passing. Since BASIC has no method for passing parameters in the subroutine call, you will have to establish your own conventions. Parameters are data for the subroutine to work with or on. Parameters (called arguments in some circles) can be input (to the subroutine) or output (values returned to the main program by the subroutine). Section VI describes one way to set up parameter passing conventions.

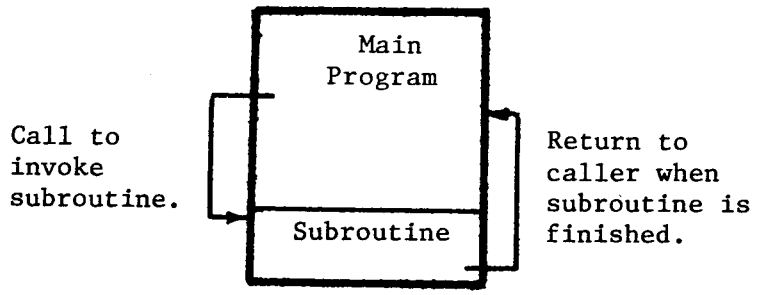


Fig. 4-1.

A. GO SUB

To get to a subroutine the statement GO SUB nnn must be executed. The statement at line number nnn is executed next. GO SUB works the same as GO TO except that GO SUB keeps around information on where it (GO SUB) was executed. This allows the subroutine to return to where it was called.

B. RETURN

The RETURN statement is used to end a subroutine and get back to where the subroutine was called.

Exercise: Show how subroutines are used.

```
T:  NEW
T:  100 PRINT "MAIN PROGRAM"
T:  110 GO SUB 1000
T:  120 PRINT "MAIN PROGRAM AFTER SUBROUTINE"
T:  130 STOP
T:  1000 PRINT "SUB 1000 CALLED"
T:  1190 RETURN

T:  RUN
R:  MAIN PROGRAM
R:  SUB 1000 CALLED
R:  MAIN PROGRAM AFTER SUBROUTINE
R:  BREAK IN 130
```

Exercise: Show what happens if a RETURN is executed with no GO SUB.

```
T:  RUN 1000
R:  SUB 1000 CALLED
R:  ?RETURN WITHOUT GOSUB ERROR IN 1190

T:  110 GO TO 1000
T:  RUN
R:  MAIN PROGRAM
R:  SUB 1000 CALLED
R:  ?RETURN WITHOUT GOSUB ERROR IN 1190
```

Exercise: Check the maximum subroutine next depth. That is, how many GOSUB's can you do without a RETURN.

```
T:  NEW
T:  5 I=-1
T:  10 I=I+1
T:  20 GOSUB 10
T:  RUN
R:  ?OUT OF MEMORY ERROR IN 10
T:  PRINT I
R:  25
```

The maximum depth of subroutine nesting that the PET allows is 25.

C. ON X GO SUB

The GOSUB statement is similar to the GOTO statement in that a single destination (line number) is specified. Just as you can make a multiway unconditional branch with the ON X GO TO nl,n2,. . . you can select one of several subroutines with ON X GO SUB nl, n2,. . .

Exercise: Show how ON X GO SUB. . . works.

```
T:  NEW
T:  100 INPUT "ENTER I";I
T:  110 ON I GO SUB 1000, 2000, 3000
T:  120 PRINT "BACK TO MAIN"
T:  130 GO TO 100
T:  1000 PRINT "FIRST (1000) SUBROUTINE"
T:  1990 RETURN
T:  2000 PRINT "SECOND (2000) SUBROUTINE"
T:  2990 RETURN
T:  3000 PRINT "THIRD (3000) SUBROUTINE"
T:  3990 RETURN

T:  RUN
R:  ENTER I?
T:  1
R:  FIRST (1000) SUBROUTINE
R:  BACK TO MAIN
R:  ENTER I?

T:  0
T:  BACK TO MAIN
R:  ENTER I?
```

D. SUBROUTINE ERROR MESSAGES

Your PET will respond with several different error messages, if you use subroutines improperly. Most of these errors were demonstrated earlier. They are summarized here to make it easier for you to find.

1. ?OUT OF MEMORY ERROR IN
2. ?RETURN WITHOUT GOSUB ERROR IN
3. ?UNDEFN'D STATEMENT ERROR
4. ?ILLEGAL QUANTITY ERROR IN

V. DATA REPRESENTATION AND PROCESSING

Before using the control statements in BASIC, some preliminary information needs to be covered. If you are already familiar with the relation operators, number systems, character codes, and logical operators, and how information is stored in memory, skip this section.

A. Information

If you understand certain basic terms and conventions that are used with digital computers, you will find it easier to understand how they operate. Common words, such as "address" and "location" take on specialized meanings when used in reference to the computer. Computer terms and conventions can be divided into two categories; units of information, and storage or retrieval of information.

1. Units of information. All information processed by a computer, whether data or instructions, can be divided into units of various sizes. The sizes range from the smallest element of information a computer can recognize to the much larger elements that the computer can manipulate. The forms of the various units of information used by computers as well as the terms used to describe these units will be described.

   A computer cannot work directly with decimal numbers, letters or special symbols such as #,-,+,%, etc. How then does the computer understand these characters since we can see that they are used for both input and output?

   The internal electronic parts of a computer have only two operating levels or states. A current is flowing or it isn't, a part is magnetized in one direction or another, a switch is either open or closed, etc. Now we can arbitrarily pick a value for each state that a computer may assume. Since there are only two states, it is customary to pick the values, or numbers, 0 and 1. All computer information whether instructions, data, numbers, alphabetic letters, or special symbols may be represented by a combination of 0s and 1s.

   Figure 5-1 shows how a specific combination of 0s and 1s can represent a specific character. The alphabetic character "M" can be represented by the states of seven electronic circuits.

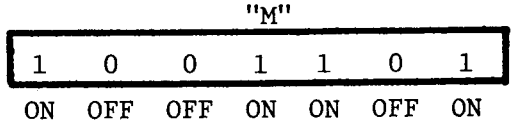
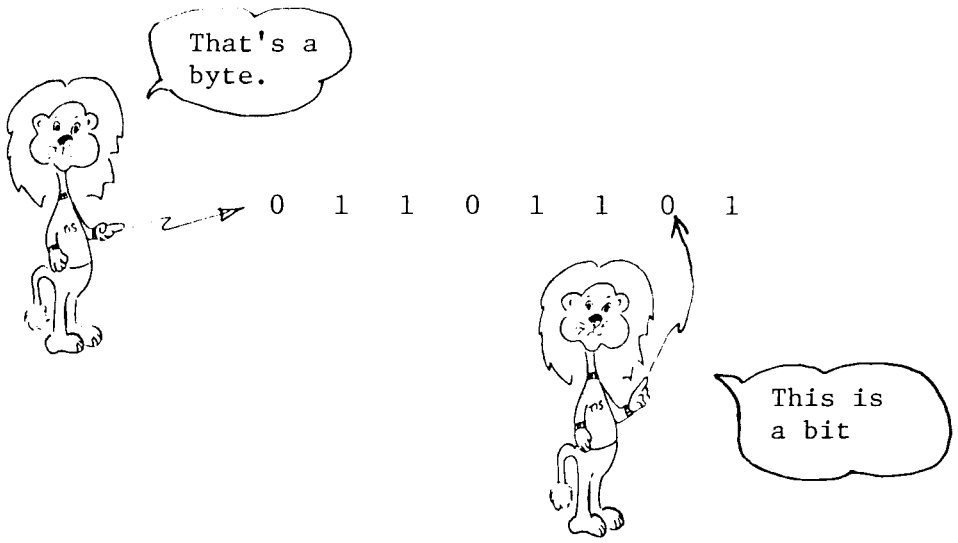


Fig. 5-1. Representing a character.

Each circuit that is on represents a 1, and each circuit that is off represents an 0. The combination shown in the figure (1001101) is a representation of the letter "M". This representation is one of convention. Since there are 128 different combinations for these seven circuits, they can collectively represent 128 unique characters.

Because a computer uses only two digits (0 and 1), they are referred to as binary digits. Thus, a binary digit, or "bit" as it is commonly called, is the smallest unit of information used in a computer. Remember a bit must always be either a 0 or a 1. Computers cannot store or retrieve individual bits when transferring information to or from memory. Therefore, larger elements of information called "bytes" or "words" are used by the computer.

Eight bits are grouped together and handled as a single element known as a "byte". A single byte might represent a decimal number, a letter, a special symbol, or part of an address. The next largest element of information is a "word." A word is made up of two or more bytes depending on the design of the computer. Most microcomputers use bytes as the largest element of information. A byte is, therefore, the smallest unit of information that is addressable, i.e., can be stored or retrieved from memory. A bit, although the smallest unit of information, is not individually addressable. Remember, a byte is 8 bits long.

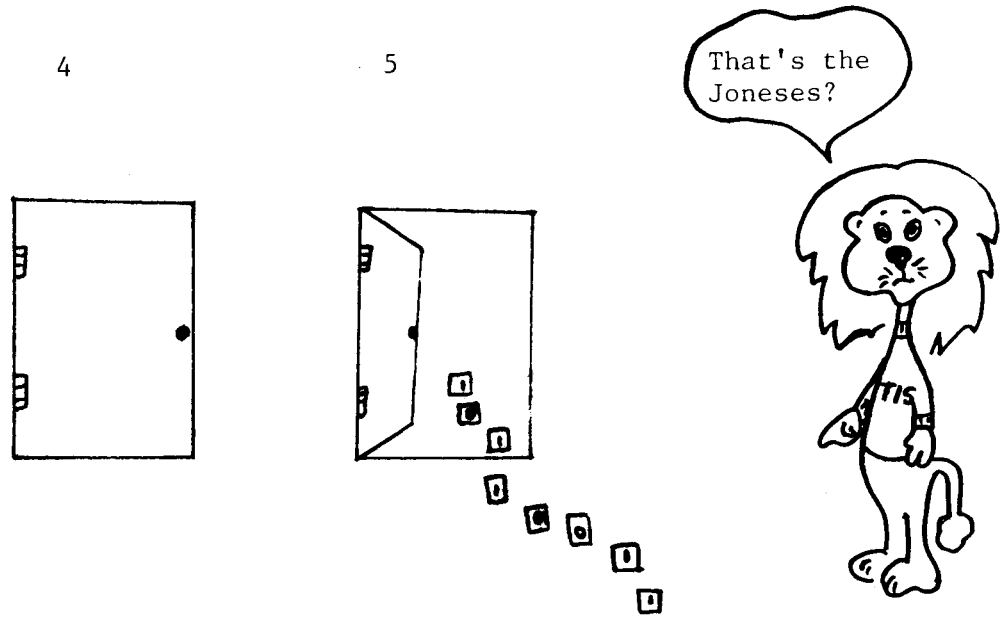


2. Storage and retrieval of information. Any information to be used by the computer must be stored where the computer can find it when needed. When specific information is needed, it must be located within the storage area and then retrieved.

Because main memory can store thousands of bits of information, the memory must be arranged in such a fashion that the central processing unit (CPU) will know where to store information and where to obtain the information it needs to perform a job.

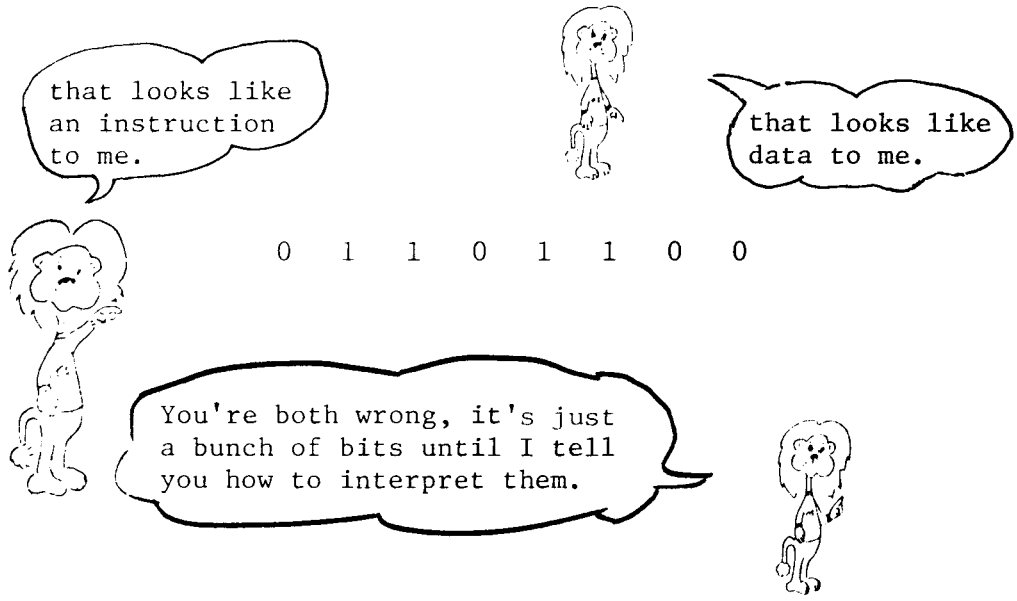
The main memory is made up of thousands of individual cells or memory locations. In our case, these are the individual bytes. Each location is used to store information that the CPU (central processing unit) will use for processing or holding the results of processing. An individual location can only hold one piece of information at a time. Each location in main memory is given a unique address at the time of manufacture. It is important to remember that the address specifies the location of a byte of memory and not the contents of that memory.

When information is stored in a location, it replaces the old contents of that location. However, when information is retrieved from a location the contents are not altered. Once information is stored, it may be read over and over. A location is like a motel room where information lives for a period of time. Since we have many motel rooms, we can assign each room a unique number or address. Now, when we have a request for a room, say by the Jones' information, we can assign the room with address 5. The contents or residents of location 5 is the Jones' information who have asked for reservations for 5 microseconds.



Memory addresses are numbers that begin at 0 and progress to the highest number required to identify all the locations within a specific memory. Because each location has its own unique address, the CPU can go directly to any location in memory. Although the address of a location is permanent, the contents of a location can be changed at any time.

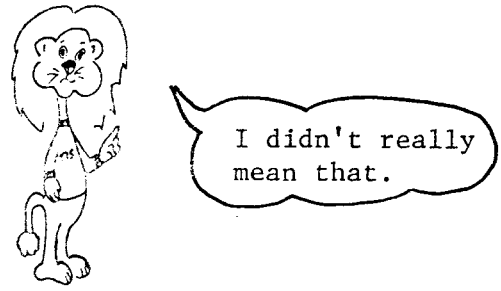
Before the computer can perform any job, it needs two types of information; the data to be processed and the instructions that tell the computer what to do with the data. Both instructions and data are stored in main memory. The only difference between instructions and data is the way the computer tries to interpret the information in a given byte.



Computer memories are available in different sizes. Memory size refers to the total number of bytes that are available. The letter "K" is normally used to indicate the size of a particular memory. "K" represents the number 1,024. Thus, a one K memory contains 1,024 bytes. It is important to remember that each memory location (byte) is assigned a unique address. A memory of 32K bytes must have 32,768 unique addresses.

B. Symbols

Symbols convey information; the symbol itself is not the information but merely represents it. The printed characters on this page are symbols and when understood convey the writer's meaning.



The meaning of symbols is really one of convention and definition. A symbol may convey one meaning to some persons, a different meaning to others, and no meaning to those persons who do not understand its significance. In order to communicate with the computer, we must use the set of symbols that the computer has been designed to accept and understand. The choice of those symbols and their meaning is a matter of convention on the part of the designers. Many of the symbols used in BASIC are the standard symbols used in everyday writing. A few symbols have been changed to accommodate the terminals that have been used and to avoid confusion with other symbols.

1. Arithmetic Symbols

Table 5-1 lists commonly used arithmetic symbols. The symbols for add, subtract, and divide are self-explanatory and are the same symbols used in normal arithmetic.

Note, however, that the symbol for the multiply operation is an asterisk rather than the conventional arithmetic (X). When dealing with computers, the convention is to use the asterisk for the multiplication symbol.

Notice also that the symbol may be used to show a number raised to a certain power. As shown in the table, 2 ↑ 4 indicates 16. Raising a number to a power is also referred to as exponentiation because it is identical to placing an exponent after the number. For instance, the expression 2 ↑ 4 can be read as either "two raised to the fourth power" or as "two with an exponent of four." In either case, the function of the exponent (or power) is the same. It indicates how many times the base number is multiplied by itself. Thus, 2 indicates that the number 2 is multiplied by itself 3 times (2 times 2 times 2 times 2).

TABLE 5-1 Arithmetic Symbols

Symbol	Meaning	Example
+	add	Y + 7
-	subtract	Y - 7
*	multiply	Y * 7
/	divide	Y / 7
↑	raise to the power of	2 ↑ 4

2. Relational Symbols

Computers are often used to compare two quantities to determine if they are equal or not equal, or to determine if one quantity is larger or smaller than the other. The symbols used to express these conditions are listed in Table 5-2

TABLE 5-2 Relational Symbols

Symbol	Meaning	Example	Comments
=	equal to	A = B	A is equal to B
<>	not equal to	A<>B	A is not equal to B
>	greater than	X > 7	X is greater than 7
<	less than	X < 7	X is less than 7
<=	less than or equal to	X<= 7	X is less than or equal to 7
>=	greater than or equal to	X>= 7	X is greater than or equal to 7

The Symbol =

In BASIC, the = sign has two interpretations. In Table 5-2 the = sign is used as a conditional symbol: A = B. This means to test for an equality condition; that is, whether the value of A precisely equals the value of B.

The second way in which the = symbol can be used is when writing equations for computer programs. In this context, the meaning of the = sign is interpreted as "is replaced by." For example, the equation, A = B, is read as "the value of A is replaced by the current value of B."

Assume that location A has a 3 stored in it, and assume location B has a 10 stored in it. The equation A = B changes only the contents of A. The location A would now have a 10 and location B would also have a 10.

As you may have noticed, we used the symbols A and B to reference two memory locations and let BASIC assign the specific addresses to A and B. This is known as symbolic addressing.

C. Number Systems

When people process information manually, they deal with alphabetic and numeric characters. However, because computers are binary machines, they represent information internally with 1s and 0s. Therefore, computers must perform mathematical operations using the binary number system and must represent alphanumeric information using binary codes.

To prepare information effectively for computer processing, people must understand how the computer will manipulate the information. This section covers binary numbers. It also presents the octal and hexadecimal number systems which many people use as a convenient, shorter form in place of binary when dealing with computer information.

The four number systems that we will work with are: the decimal system, which we use in our everyday work; the binary system, which digital computers use because it is easy to implement electronically; and the octal and hexadecimal systems, which programmers use to represent binary numbers because they are easier to work with. Hexadecimal (or hex for short) and octal numbers, however, must still be converted to binary before the computer can process them.

As shown in Table 5-3, each number system has a unique base or radix. This base corresponds to the number of digits or unique symbols that are used in that number system. For example, the decimal number system uses 10 digits, 0 through 9, and therefore has a radix of 10.

TABLE 5-3 Actual Values

NUMBER SYSTEM	BASE (RADIX)	ACTUAL VALUES (DIGITS)
Decimal	10	0 through 9
Hexadecimal	16	0 through 9 and A - F
Octal	8	0 through 7
Binary	2	0 and 1

1. Decimal - Binary - Hexadecimal

In the decimal system, two or more digits are needed to represent any value greater than 9. Whenever a number contains two or more digists, each digit is assigned a specific value called a place or positional value. This place value equals the base of the number system raised to some power. See section V - B-1 (Arithmetic Symbols) for an explanation of power. In the decimal system, place values are based on powers of ten (that is, units, tens hundred, thousands, etc.)

Decimal to Binary Conversion

When converting a decimal number to a binary number, the following rules apply:

a. Divide the decimal number by 2 and save the remainder.
b. Divide the quotient from the previous division by 2 and save the remainder.
c. Continue step b until the quotient is zero.
d. The remainders saved from the division make up the digits of the binary number. The first remainder is the least significant digit (LSD), and the last remainder is the most significant digit (MSD).

The example below shows how the decimal number 38 is converted to binary.

BASE	NUMBER	QUOTIENT	REMAINDER
2	38	19	0 (LSD)
2	19	9	1
2	9	4	1
2	4	2	0
2	2	1	0
2	1	0	1 (MSD)

Therefore: 38 = 1 0 0 1 1 0

Exercise: Write a program to display the binary representation of a decimal program number.

```
T:  NEW
T:  90 REM DISPLAY BINARY VALUE
T:  100 INPUT "ENTER DECIMAL WHOLE NUMBER"; R
T:  430 R$=""
T:  440 GOSUB 2000
T:  500 PRINT "DECIMAL";R; "IS BINARY ";R$
T:  510 GO TO 100
```

Main program prompts for and gets the decimal number. The subroutine at 2000 does the actual work. The main program then displays the result. The main program passes the number to convert to the subroutine in the variable R and receives the converted result in the variable R$.

```
T:  2000 REM CONVERT NUMBER TO BINARY CHARACTER
T:  2010 C$(0)="0":C$(1)="1"
T:  2020 R2=R
T:  2030 T2% R2/2
T:  2040 S=R2-T2%*2
```

S is the remainder and T2% is the quotient.

```
T:  2050 R$=C$(S)+R$
T:  2060 R2=T2%
T:  2070 IF T2%  OTHEN 2030
```

The old quotient (T2%) becomes the new dividend (R2). Repeat until all binary digits have been assembled in R$.

```
T:  2080  RETURN
```

Save this program as DEC-BIN. It will be used later to show how logic operations are performed.

```
T:  RUN
R:  ENTER DECIMAL WHOLE NUMBER
T:  38
R:  DECIMAL 38 IS BINARY 1 0 0 1 1 0
R:  ENTER DECIMAL WHOLE NUMBER?
```

Try a few decimal numbers. You can use this program to check if you are able to successfully convert from decimal to binary by hand.

```
T:
R:
R:  ENTER DECIMAL WHOLE NUMBER?
T:
R:
```

To terminate this program, type 'STOP' and then 'RETURN'.

In the binary, or base-two number system, we have the two unique symbols 0 and 1. The place value in this system is a power of two, that is, units, twos, fours, eights, sixteens, etc. Now let's see if Linus can illustrate this

First let's number some boxes beginning with 0.

5 4 3 2 1 0

These numbers are going to be used to determine the place value of whatever number system we choose to use.

Why did we start with the number 0?

Because we need a units position and any number raised to the Zero power is 1.

$2 \uparrow 0 = 1$

To determine the value of a number, you must first know in what base it is represented. You then take this base and, using the numbers in the above box as exponents, you determine the place values. Next, you multiply each digit in the number by its corresponding place value. Last, you add all these numbers together. Let's determine the value of the binary number 1011. First, let's recreate the box to determine the exponents.

5 4 3 2 1 0

2↑0 = 1

2↑1 = 2

2↑2 = 4

2↑3 = 8

2↑4 = 16

2↑5 = 32

See how we use the numbers in the box.

The base is two (binary)

Those are the place values.

5 4 3 2 1 0

32 16 8 4 2 1

Multiplying each digit of 1011 by its place value

1 X 1 = 1

1 X 2 = 2

0 X 4 = 0

1 X 8 = 8

Now adding 8 + 0 + 2 + 1 = 11

The value of the binary number 1011 is equal to decimal 11.

The above method works with any number system.

Exercise: Convert a binary number in a character string form to a decimal number. Load the DEC-BIN program.

```
T:  90 REM DISPLAY DECIMAL VALUE
T:  100 INPUT"ENTER BINARY NUMBER";N$
T:  110 T1$=N$
T:  120 GOSUB1100
T:  130 IFT<0THEN100
T:  140 PRINTN$;"BINARY IS";T;"DECIMAL"
T:  150 GOTO100
```

The main program gets the character string with the binary number and passes it to the conversion subroutine (at 1100). If the conversion is successful, the value is printed out.

```
T:  1100 REM CONVERT BINARY CHARACTER STRING TO DECIMAL NUMBER
T:  1105 T=0
T:  1110 L1=LEN(T1$)
T:  1120 FORI1=1TOL1
T:  1125 T=T*2
T:  1130 IFMID$(T1$,I1,1)="1"THENT=T+1:GOTO1150
T:  1140 IFMID$(T1$,I1,1)<>"0"THENT=-1:RETURN
T:  1150 NEXTI1
T:  1160 RETURN
```

The subroutine converts the binary character string to a decimal number. If any character besides 0 or 1 is found an error flag is returned.

```
T:  RUN
R:  ENTER BINARY NUMBER
T:  101
R:  101 BINARY IS 5 DECIMAL
```

Try a few binary numbers. Save this program as BIN-DEC. We will add to it later.

The hexadecimal number system, sometimes referred to as "hex," has a base of sixteen. A base of sixteen means that the system uses sixteen unique symbols. The symbols that are commonly used are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Table 5-4 illustrates the counting numbers for four different number systems.

TABLE 5-4

DECIMAL Base 10	HEXADECIMAL Base 16	OCTAL Base 8	BINARY Base 2
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	8	10	1000
9	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	E	16	1110
15	F	17	1111

Notice that it takes more digits to represent a number if the base is small. For example, the number 15 requires 4 binary digits, 2 octal digits, 2 decimal digits and only 1 hexidecimal digit.

Let's convert the hex number 8AF to decimal. Remember that any base raised to the zero power is always one, and any base raised to the first power is the base itself.

16 0 = 1

16 1 = 16

16 2 = 256

These are the place values.

F X 1 = 15 X 1 = 15

A X 16 = 10 X 16 = 160

8 X 256 = 2048

F hex is equal to 15 decimal

Now add the results

15 + 160 + 2048 = 2223

The hex number 8AF = 2223 decimal

2. Conversions

It is often necessary to convert from one number system to another. The four conversion techniques covered here are:

a. Binary to Octal
b. Binary to Hexadecimal
c. Binary to Decimal
d. Decimal to Binary

Binary to Octal Conversion:

Conversion between octal and binary numbers is fairly straightforward. 8 decimal = 10 octal = 2 ↑ 3. This means that every octal digit is equivalent to three binary digits and vice versa. See Table 5-5.

TABLE 5-5 Octal-Binary Equivalents

OCTAL	BINARY
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

The conversion procedure is as follows:

1. Start at the right hand side and arrange the binary digits into groups of three.

2. Substitute the equivalent octal digit from table 5-5.

Let's convert the binary number 110101111001 to octal.

110	101	111	001	Separate into groups of three.
6	5	7	1	Substitute the octal digits from the table.

$110101111001_2 = 6571_8$

Binary to Hexadecimal Conversion:

Conversions between hex and binary is also straight forward since 16 is another power of 2; namely 16 = 2 4. This means that you can use the same procedure you used above except that you arrange the binary number into groups of four. Let's use Table 2 and convert the same binary number (1101011111001) into hex.

TABLE 5-6 Hexadecimal-Binary Equivalencies

HEXADECIMAL	BINARY
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

1101	0111	1001	Separate into groups of four.
D	7	9	Substitute the hex digit from the table.

$110101111011_2 = D79_{16}$

D. Character Codes

The method or system used to represent data is known as a code. In the computer, the code relates data to a fixed number of bits. If you use only a single bit to represent each symbol, you are limited to two choices: one symbol for the "ON" state and one symbol for the "OFF" state. This would severely limit our ability to communicate. If you add another bit, you double the number of characters that can be represented. You have two choices with one bit; 0 or 1. You have four choices with two bits; 00, 01, 10, or 11. If you add another bit you again double the number of choices; 000, 001, 010, 011, 100, 101, 110, and 111. It can be shown that for n bits the number of choices, or characters is $2 \uparrow n$.

The American Standard Code for Information Interchange (ASCII) is a 7-bit code that has been established as a standard by the computer industry. The PET uses a modified version of ASCII.

E. Logical Operations

The three logical functions that are available in PET BASIC are

a) AND
b) OR
c) NOT

Each of these functions, except NOT has two input values but only one output value (result). The NOT function has one input value and one output value. These values are normally referred to as "true" and "false". The PET uses non-zero for true and zero for false.

Exercise: Determine what numeric values the PET uses to represent true and false.

```
T:  NEW
T:  100 PRINT "ENTER VALUE TO TEST TRUE/FALSE
T:  110 INPUT A
T:  200 IF A THEN PRINT "A IS TRUE WHEN A=";A: GOTO 100
T:  210 PRINT "A IS FALSE WHEN A=";A: GOTO 100
T:  RUN
R:  ENTER VALUE TO TEST TRUE/FALSE
T:  1
R:  A IS TRUE WHEN A = 1
R:  ENTER VALUE TO TEST TRUE/FALSE
T:  0
R:  A IS FALSE WHEN A = 0
R:  ENTER VALUE TO TEST TRUE/FALSE
T:  .001
R:  A IS TRUE WHEN A = 1E - 3
R:  ENTER VALUE TO TEST TRUE/FALSE
T:  1
R:  A IS TRUE WHEN A =-1
R:  ENTER VALUE TO TEST TRUE/FALSE
```

These results show that the PET interprets zero as false and non-zero as true. However, not just any non-zero value will work properly for true. When NOT is covered, you will find that true must be represented with the non-zero value -1.

Exercise: Determine what string values the PET uses to represent true and false.

```
T:  NEW
T:  100 PRINT"ENTER STRING TO TEST TRUE/FALSE"
```

```
T:  110 INPUT A$
T:  200 IF A$ THEN PRINT "A$ IS TRUE WHEN A$=";A$:GO TO 100
T:  210 PRINT "A$ IS FALSE WHEN A$=";A$: GO TO 100
T:  RUN
R:  ENTER STRING TO TEST TRUE/FALSE
T:  ABC
R:  A$ IS TRUE WHEN A$=ABC
R:  ENTER STRING TO TEST TRUE/FALSE
T:  ""
R:  A$ IS FALSE WHEN A$=
R:  ENTER STRING TO TEST TRUE/FALSE
T:  "b"
R:  A$ IS TRUE WHEN A$=
R:  ENTER STRING TO TEST TRUE/FALSE
```

This shows that the null string ("") is interpreted as false. Any string with at least one character in it is considered true.

1. AND Function

The AND Function gives a true output only when both inputs are true.

The statement:

Tim AND Bill own computers; is true only if both Tim and Bill do own computers.

A chart can be built to describe the operation of any logic function. This chart lists every possible combination of input values (binary 1s and 0s) and the output resulting from each combination of inputs. Some people substitute the words "true" and "false" for the values "1" and "0" when using a chart of this type. That's why this chart, or table, is commonly referred to as a "truth table" even though 1s and 0s are often used.

Table 5-7 is a truth table for a 2-input AND function. It lists every possible combination of input values and shows the appropriate output for each combination of inputs. Notice that there is an output only when both inputs are true (or 1s). Thus, we can say that "C" (the output) is true when inputs "A" AND "B" are true.

TABLE 5-7 2-Input AND Function - Truth Table

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Another method of describing the AND function is to use a mathematical formula. Using a formula provides a precise shorthand notation for easy representation of digital logic. Let's go back to the 2-input AND function. You know that the output ("C") is true only when both inputs ("A" and "B") are true. Therefore, you can express this relationship by using the formula:

C = A AND B

Exercise: Display AND Truth Table.

```
T:  NEW
T:  90  PRINT "INPUT OUTPUT"
T:  95  PRINT " A  B  C"
T:  100 FOR I= 0 TO 1
T:  110 FOR J - 0 TO 1
T:  120 PRINT I;J;I AND J
T:  130 NEXT J
T:  140 NEXT I
T:  RUN
R:                          TABLE 5-7 should be displayed.
```

2. Inclusive OR function

The inclusive OR gate provides a true output if either one input or the other input or both inputs are true. This function produces a true output (binary 1) when at least one input logic condition is satisfied as indicated by a binary 1 on one of the input values. Thus, we can say that 1 OR 1 = 1.

The statement:

Don OR Dave can program is true if either Don or Dave (or both) do know how to program .

This function is called an inclusive OR because any one or more true inputs will provide a true output. In other words, if there were four input values, a true input on any one of the values 1 through 4, inclusive, would provide a true output. Whenever the term "OR" is used by itself, it refers to the "inclusive OR".

Table 5-8 is a truth table for a 2-input inclusive OR function. It lists all possible combinations of input values and shows the output for each combination of inputs.

TABLE 5-8 2-Input OR - Truth Table

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

A mathematical formula can also be used to represent the inclusive OR function. For example, operation of a 2-input inclusive OR can be expressed by the formula:

C = A OR B

Exercise: Display OR Truth Table. Use the program for AND with one modification.

```
T: 120 PRINT I;J; I OR J
T: RUN
R:
```

TABLE 5-8 should be displayed.

3. NOT Function (Inverter)

The purpose of the NOT function is to convert an input value to its opposite (complementary) state. The NOT function is often called an inverter because it inverts the input value. For example, if the input signal is one, the NOT inverts it to a zero.

The statement:

Dick is NOT an engineer is false if he is an engineer and true otherwise.

Table 5-9 is a truth table for the NOT function. Notice that there are only two possible inputs to the function.

TABLE 5-9 NOT function - Truth Table

INPUT	OUTPUT
0	1
1	0

We can also use a mathematical formula to represent the NOT function. We express the NOT function as

C = NOT A

Exercise: Display NOT Truth Table. It may seem reasonable to simply modify the program for the AND and the OR truth tables. This will not work because the NOT truth table (Table 5-9) shows that NOT true must be false and NOT false must be true.

You have been using 1 for true and 0 for false. Since NOT false should be true try:

```
T:  PRINT NOT 0
R:  -1
```

Not 0 is -1. Therefore, you cannot use 1 for logical value true. To double check, NOT true must be false.

```
T:  PRINT NOT -1
R:  0
```

The PET uses zero to represent false and -1 to represent true.

```
T:  NEW
T:  100 FOR I=0 TO -1 STEP -1
T:  110 PRINT ABS(I); ABS(NOT I)
T:  120 NEXT I
T:  RUN
R:
```

TABLE 5-9 should be displayed.

In the section on memory organization you learned that the computer cannot manipulate individual bits separately. The smallest addressable unit is a byte. The logical operations work on each bit in the 8-bit byte in parallel. The PET treats 2 eight bit bytes as the unit for logical operations.

Exercise: Determine the maximum size of quantities used in logical operations.

```
T:  NEW
T:  A=1:B=2
T:  X=32767: PRINT (A<B) AND X
R:  32767
T:  X = X*2+1: PRINT (A<B) AND X
R:  ?ILLEGAL QUANTITY ERROR
```

Logical operations are only permitted on 16 bit (or less) quantities.

Exercise: Display the value of the logical function performed on various data. Load the program BIN-DEC and add the following lines: (A complete listing appears in the Appendix.)

```
T:  140 NE=T
T:  150 INPUT "ENTER LOGICAL FUNCTION (A,O,N)";OP$
T:  190 F$=LEFT$(OP$,1)
T:  200 IF F$  "A" AND F$  "O" AND F$  "N" THEN 150
```

Get the logical operator (AND, OR, NOT) and check if the first character is correct.

```
T:  210 TW$="b"
T:  220 IF F$ = "N" THEN 420
```

Since NOT needs only 1 argument, skip past input of and conversion of second argument.

```
T:  300 "ENTER SECOND BINARY NUMBER";TW$
T:  310 T1$=TW$
T:  320 GOSUB1100
T:  330 IF T< O THEN 300
```

Make sure the number entered is a legal binary value.

```
T:  340 TWO=T
T:  400 IF F$="A" THEN R=NE AND TWO
T:  410 IF F$="O" THEN R=NE OR TWO
T:  420 IF F$="N" THEN R=NOT NE
```

Perform the requested logical operation on the input parameter(s).

```
T:  500 PRINT N$, F$, TWO$, "IS",R$
```

Print the input and operation with the result.

Save this program on a cassette. Call it LOGIC. Then try some test data.

```
T:  RUN
R:  ENTER BINARY NUMBER
T:  101
R:  ENTER LOGICAL FUNCTION (A,O,N)
T:  A
R:  ENTER SECOND BINARY NUMBER
T:  11
R:  101 A 11 IS 1
R:  ENTER BINARY NUMBER
```

Try a few more values and operations to check the program.

F. Relational Operations

The relational operations determine if the relation holds and assigns the value true (-1) or false (0).

Exercise: Determine what the relational operators generate as a result.

```
T:  A=1: B=2: X=A<B: PRINT X
R:  -1
T:  PRINT A>B
R:  0
T:  PRINT A<B
R:  -1
T:  PRINT A=B
R:  0
```

Using -1 for true and 0 for false shows that the result of a relational operation is the logical value of the relation.

VI. BUILDING YOUR OWN LIBRARY

After you have written several programs, you will notice that there are similar tasks to handle in every program. Once you have developed a technique to solve a particular problem, you can use the same technique later. The best way to reuse pieces of programs is to make those pieces into subroutines. Then, whenever you need a particular task performed, you only need to GOSUB to the subroutine.

The biggest problem with BASIC subroutines is the fact that all variables are global, i.e., any subroutine can use (change) any variable. This is OK and even useful if the variable change was expected. However, if the variable was changed unexpectedly, you have a "bug".

There are two types of variables used with subroutines. There are parameters for the subroutine to act on and local (temporary) variables needed by the subroutine to perform the action. What is needed is a variable naming system that will insure that there will be no conflicts between main program and subroutine.

Since subroutines can only have numbers (not names), you can use the numbers to construct variable names. Let's assume that you have designed a useful subroutine. The subroutine needs two input arguments and two results returned to the caller. In addition, it needs three temporary variables. How can you organize your subroutines so that there will be no conflicts?

You should set up a naming convention to keep track of which subroutines use which variables. The hardest bugs to track down are those that are caused by adverse side effects. An adverse side effect of an otherwise working section of code can make another (different) section of code stop working properly.

If you layer your subroutines in a hierarchy based on the statement number range, you can use the most significant digit of the statement number as part of the naming convention. The second step is to be sure in your "layering" that subroutines only call subroutines at a lower (more subservient) level.

Example: Set up a set of variable naming conventions. In this example n stands for the most significant digit in the statement number.

A. Parameter Passing

Variable Names	Quantity Stored in Variable
Rn$	READ file name
Wn$	WRITE file name
Sn$	SAVE file name
Vn$	VERIFY file name
Rn%	READ file number
Rn	READ file status
Wn%	WRITE file number
Wn	WRITE file status

Variable Names	Quantity Stored in Variable
An$	Single string "accumulator"
An	Single floating point "accumulator"
An%	Single interger "accumulator"
Bn$	Single secondary string "accumulator"
Bn	Single secondary FP "accumulator"
Bn%	Single secondary interger "accumulator"
Cn$	Single third character "accumulator"
Cn	Single third FP "accumulator"
Cn%	Single third interger "accumulator"

B. Temporary Locations

In, Jn, Kn, Ln, In%, Jn%, Kn%, Ln% — Temporary for loop indices

En$, En, En% — Single error flag or status indications

Fn$, Fn, Fn% — Single flags

Tn$, Tn, Tn%, Un$, Un, Un%, Xn$, Xn, Xn%, Yn$, Yn, Yn%, Zn$, Zn, Zn% — Temporary single variable storage

Results can either be returned in the accumulator(s) or in result registers.

Variable Names	Quantity Stored in Variable
Qn$ Qn Qn% Pn$ Pn Pn% Nn$ Nn Nn%	Single variable returned values

Using return registers for results can make debugging simpler.

After calling a subroutine, you can display what went in, the latest value of temporaries, and the results generated by the subroutine.

If you use the accumulators to return results, you destroy the original contents of the accumulator(s) so you can't see what the subroutine was called to work on. However, using the accumulator(s) to return results allows you to string together subroutines without having to move results to accumulators between subroutine calls.

These naming conventions gives you the following number of variable names of each type for use in subroutines:

3 Input accumulators
3 Output (result) registers
4 Temporary FOR loop indices
1 Error indicator
1 Flag register
5 Temporary variables
1 Read file name (number
1 Write file name (number)
1 Save file name (number)
1 Verify file name (number)

Since there are 6 types, (string, floating point, and integer for both single and array reference) you can multiply each of the counts by 6. That should give you more variable names of each kind than you will ever need.

You still have left all of the two alphabetic character variable names for your main program.

Since the PET treats array variable names separately from single variables, you can use the same names (subscripted) where you need arrays.

APPENDIX PROGRAM LISTINGS

Complete listing of the final version of the mailing list skeleton program (MAIL4)

```
100 PRINT"MAILING LIST SKELETON"
102 AT$(1)="ADD"
104 AT$(2)="CHANGE"
106 AT$(3)="PRINT"
108 AT$(4)="COUNT"
110 INPUT"ENTER ACTION ";A$
115 FORA=1TO4
120 IFA$<>AT$(A)THEN135
130 ON A GOTO 1000,2000,3000,4000
135 NEXTA
140 PRINT"INPUT MUST BE LEGAL ACTION"
150 GOTO110
1000 PRINT"ADD SELECTED"
1990 GOTO150
2000 PRINT"CHANGE SELECTED"
2990 GOTO150
3000 PRINT"PRINT LIST REQUESTED"
3990 GOTO150
4000 PRINT"COUNT REQUESTED"
4990 GOTO150
```

Program to convert decimal numbers to binary character string and display it.

```
90 REM DISPLAY BINARY VALUE
100 INPUT"ENTER DECIMAL WHOLE NUMBER";R
440 GOSUB2000
500 PRINT"DECIMAL";R;"IS BINARY ";R$
510 GOTO100
2000 REM CONVERT NUMBER TO BINARY CHARACTER STRING
2010 C$(0)="0":C$(1)="1"
2020 R2=R
2025 R$=""
2030 T2%=R2/2
2040 S=R2-T2%*2
2050 R$=C$(S)+R$
2060 R2=T2%
2070 IFT2%<>0THEN2030
2080 RETURN
```

Program to convert binary character string to a decimal number and display it.

```
90 REM DISPLAY DECIMAL VALUE
100 INPUT"ENTER BINARY NUMBER";N$
110 T1$=N$
120 GOSUB1100
130 IFT<0THEN100
140 PRINTN$;"BINARY IS";T;"DECIMAL"
150 GOTO100
440 GOSUB2000
500 PRINT"DECIMAL";R;"IS BINARY ";R$
510 GOTO100
1100 REM CONVERT BINARY CHARACTER STRING TO DECIMAL NUMBER
1105 T=0
1110 L1=LEN(T1$)
1120 FORI1=1TOL1
1125 T=T*2
1130 IFMID$(T1$,I1,1)="1"THENT=T+1:GOTO1150
1140 IFMID$(T1$,I1,1)<>"0"THENT=-1:RETURN
1150 NEXTI1
1160 RETURN
2000 REM CONVERT NUMBER TO BINARY CHARACTER STRING
2010 C$(0)="0":C$(1)="1"
2020 R2=R
2025 R$=""
2030 T2%=R2/2
2040 S=R2-T2%*2
2050 R$=C$(S)+R$
2060 R2=T2%
2070 IFT2%<>0THEN2030
2080 RETURN
```

Complete listing of the display logical function program (LOGIC)

```
90 REM DISPLAY LOGICAL FUNCTIONS
100 INPUT"ENTER BINARY NUMBER";N$
110 T1$=N$
120 GOSUB1100
130 IFT<0THEN100
140 NE=T
150 INPUT"ENTER LOGICAL FUNCTION (A,O,N)";OP$
190 F$=LEFT$(OP$,1)
200 IFF$<>"A"ANDF$<>"O"ANDF$<>"N"THEN150
210 TW$=" "
220 IFF$="N"THEN420
300 INPUT"ENTER SECOND BINARY NUMBER";TW$
310 T1$=TW$
320 GOSUB1100
330 IFT<0THEN300
340 TWO=T
400 IFF$="A"THEN R=NE AND TWO
410 IFF$="O"THEN R=NE OR TWO
420 IFF$="N"THEN R=NOT NE
430 R$=""
440 GOSUB2000
500 PRINTN$;" ";F$;" ";TWO$;" IS ";R$
510 GOTO100
1100 REM CONVERT BINARY CHARACTER STRING TO DECIMAL NUMBER
1105 T=0
1110 L1=LEN(T1$)
1120 FORI1=1TOL1
1125 T=T*2
1130 IFMID$(T1$,I1,1)="1"THENT=T+1:GOTO1150
1140 IFMID$(T1$,I1,1)<>"0"THENT=-1:RETURN
1150 NEXTI1
1160 RETURN
2000 REM CONVERT NUMBER TO BINARY CHARACTER STRING
2010 C$(0)="0":C$(1)="1"
2020 R2=R
2025 R$=""
2030 T2%=R2/2
2040 S=R2-T2%*2
2050 R$=C$(S)+R$
2060 R2=T2%
2070 IFT2%<>0THEN2030
2080 RETURN
```